Software Qual J (2017) 25:803–840
DOI 10.1007/s11219-016-9339-1

Separating passing and failing test executions by clustering anomalies

Rafig Almaghairbe[1] · **Marc Roper**[1]

Published online: 3 October 2016

Abstract Developments in the automation of test data generation have greatly improved efficiency of the software testing process, but the so-called oracle problem (deciding the pass or fail outcome of a test execution) is still primarily an expensive and error-prone manual activity. We present an approach to automatically detect passing and failing executions using cluster-based anomaly detection on dynamic execution data based on firstly, just a system's input/output pairs and secondly, amalgamations of input/output pairs and execution traces. The key hypothesis is that failures will group into small clusters, whereas passing executions will group into larger ones. Evaluation on three systems with a range of faults demonstrates this hypothesis to be valid—in many cases small clusters were composed of at least 60 % failures (and often more). Concentrating the failures in these small clusters substantially reduces the numbers of outputs that a developer would need to manually examine following a test run and illustrates that the approach has the potential to improve the effectiveness and efficiency of the testing process.

Keywords Software testing · Test oracles · Anomaly detection · Clustering

1 Introduction

One of the significant recent advances in software testing has been in the area of automation. It is now possible to automatically generate test data for an arbitrary system that achieves a remarkably high level of coverage. However, there is a downside to this—unless they are specified completely and explicitly, the outputs from all this automatically

✉ Marc Roper
marc.roper@strath.ac.uk

Rafig Almaghairbe
rafig.almaghairbe@strath.ac.uk

[1] Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK

generated data need to be checked to determine their correctness or otherwise. In theory, this should just be a matter of comparing the results with those defined by the complete (and, ideally, machine processable) system specification, but unfortunately such an artefact rarely exists. As a result, a huge amount of human effort is needed if there is a large set of test cases. Involving people in the process is expensive and possibly error-prone and therefore some other strategy for building an oracle—a mechanism for determine the (in)correctness of outputs associated with inputs—needs to be developed.

Anomaly detection is a general set of strategies that can be used to detect unusual values or outliers in large data sets. It has been employed successfully in various research areas such as cyber-intrusion detection, fraud detection, industrial damage detection, image processing, system health monitoring, event detection in sensor networks, and detecting eco-system disturbances (Chandola et al. 2009). The aim of the work reported in this paper to investigate whether software bugs generate an anomalous pattern of behaviour that can be distinguished from normal (non-buggy) behaviour. If this is the case, then the possibility of detecting bugs automatically can be raised.

This paper reports on an extended experiment into the use of a range of clustering-based anomaly detection techniques to support the construction of a test oracle. In the first part of the study, a range of clustering algorithms are applied to just the test case input/output pairs of three systems and the effectiveness of this approach is evaluated. In the second part of the study the test case input/output pairs are augmented with their associated execution traces with the aim of improving the accuracy of the approach and the results of a second experiment investigating and evaluating this are presented.

The main findings of this first study provided evidence to support the feasibility of using anomaly detection techniques to separate passing and failing test results and confirm the findings of our preliminary study in the area (Rafig and Roper 2015). The results vary between systems in particular and algorithms to a lesser extent, but show that smaller than average-sized clusters exhibit both a far higher density of failures and a sample of the range of faults in the program. The practical implications of the approach suggest that the task of checking outputs can be reduced to a fraction of that normally required: the majority of failures in a system can be observed from inspecting a minority of the results. Reducing the proportion of failures in the output also demonstrates that the approach is robust to a decline in the failure intensity rate.

2 Principles and an illustrative example

The concept behind the approach explored in this paper is illustrated in Fig. 1. The Software Under Test (SUT) is executed with test case inputs which generate corresponding outputs. The paths taken by the test cases through the SUT are represented by the wavy lines in the figure, and some of these might encounter faults in the software (represented by * symbols) which may cause some of the outputs to fail. It is also possible to collect this trace information as part of this testing process. So after running the tests we have a set of input test cases, outputs from the tests, and (if we chose to collect them) execution traces. At this stage it is unknown which of the tests have passed and which have failed unless all of the outputs are examined to see if the results are as expected. The approach presented in this paper explores the application of clustering to separate the passing and failing tests into distinct clusters: the failing outputs (being less frequent) gathering in the smaller clusters and the more frequent passing tests grouping into larger clusters. Checking the results then

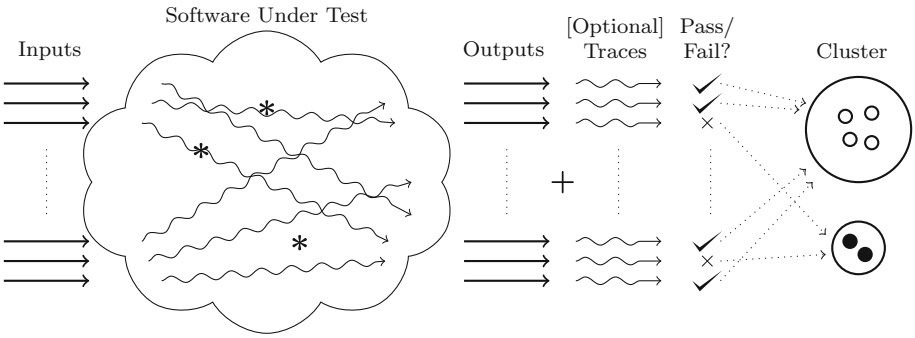


Fig. 1 Principles of using clustering to automatically classify failing outputs. The program under test is run on a set of inputs which will generate outputs and optional traces, and may encounter bugs in the program (the *'s). The pass/fail status of the outputs is unknown and the aim is to automatically separate these using clustering strategies

proceeds by examining the contents of the smallest clusters first as these should be more likely to contain the failing outputs. In this way, failing outputs are identified sooner and the process of checking results becomes more efficient.

To illustrate this idea, a small example is presented which demonstrates the principles behind the approach and the potential benefits the it offers to the software engineer. This illustration is taken from Defects4J[1]—a collection of open-source systems, faults, and an infrastructure for running and profiling tests. Part of the tutorial documentation for Defects4J contains the infamous triangle problem—a well-known testing example which takes three integer inputs and returns the corresponding type of triangle represented by these three values: equilateral, isosceles, scalene or invalid. This program comes with several faults in the form of mutants that may be applied and 35 test cases. In the example here the fault takes the form of the line in the program containing:

```
...
else if (trian == 3 && b + c > a)
   return TriangleType.ISOSCELES;
...
```

being replaced by:

```
...
else if (trian == 3)
   return TriangleType.ISOSCELES;
...
```

The tester is of course unaware of this and runs the tests to produce the results shown in Table 1. In this illustration no use is made of JUnit[2] or any other unit testing framework in order to specify the expected results. Although this is not necessarily good practice, it is not unusual: the ISTQB Worldwide Software Testing Practices Report surveyed 3200 test managers and technical staff from 89 countries and found that unit testing tools were employed in just under 43 % of organisations (ISTQB 2016). Furthermore, it is not always

[1] https://github.com/rjust/defects4j.

[2] `junit.org`.

Table 1 Inputs and actual outputs for the triangle example

Test ID	Input1	Input2	Input3	Output
T1	0	1301	1	INVALID
T2	1108	1	1	INVALID
T3	1	0	−665	INVALID
T4	1	1	0	INVALID
T5	582	582	582	EQUILATERAL
T6	1	1088	15	INVALID
T7	1	2	450	INVALID
T8	1663	1088	823	SCALENE
T9	1187	1146	1	INVALID
T10	1640	1640	1956	ISOSCELES
T11	784	784	1956	INVALID
T12	1	450	1	INVALID
T13	1146	1	1146	ISOSCELES
T14	1640	1956	1956	ISOSCELES
T15	−1	1	1	INVALID
T16	1	−1	1	INVALID
T17	1	2	3	SCALENE
T18	2	3	1	SCALENE
T19	3	1	2	SCALENE
T20	1	1	2	INVALID
T21	1	2	1	INVALID
T22	2	1	1	INVALID
T23	1	1	1	EQUILATERAL
T24	0	1	1	INVALID
T25	1	0	1	INVALID
T26	1	2	−1	INVALID
T27	1	1	−1	INVALID
T28	0	0	0	INVALID
T29	3	2	5	SCALENE
T30	5	9	2	INVALID
T31	7	4	3	SCALENE
T32	3	8	3	INVALID
T33	7	3	3	INVALID
T34	1108	1	1	ISOSCELES
T35	1108	2	2	ISOSCELES

easy to specify the results of a test (resulting in partial or incomplete oracles), and testing is also carried out at many levels: integration, system, acceptance, regression etc., where the tests may not be defined using one of the *n*Unit family of frameworks.

In typical circumstances, the tester would then proceed to work through the outputs one-by-one to check whether the test passed or failed. Instead of doing this, we firstly group the related inputs and outputs together into 35 vectors (+<0,1301,1,INVALID>,<1108,1,1,INVALID> ...<1108,2,2,ISOSCELES>+) and then apply a

clustering algorithm. This groups the data into 4 clusters, illustrated by Fig. 2, one large cluster containing 29 items and 3 much smaller ones containing 1, 2 and 3 items, respectively. By concentrating first of all on the small clusters, the tester would find two failing outputs after examining just six results: these are T34 and T35 which appear in cluster 3 along with the passing case T13 (for information cluster 1 contains T8 and cluster 2 contains T10 and T14). At this point the programmer may feel that they have enough evidence that the program is not working and choose to stop examining test results and work on debugging the program. This evidence has been obtained after looking at just a fraction of all test outputs, saving the developer time and making the testing process much more efficient.[3]

The purpose of the work reported in this paper is to explore whether this approach scales up to larger systems where there are hundreds or thousands of test cases: do failing outputs tend to gather in the smaller clusters meaning that developers can confidently focus their efforts on just a small proportion of test results, and in the case where there are multiple failures then what proportion of these feature in the small clusters?

3 Background and related work

The automatic generation of test oracles is an important problem in software testing area, but this problem has received considerably less attention compared to other testing problems such as the generation of test cases. Three extensive reviews of test oracles exist: by Baresi and Young (2001), by Pezzè and Zhang (2005), and by Barr et al. (2015) who classified the existing literature on test oracles into three broad categories:- specified oracles; implicit oracles; and derived oracles. Specified oracles are test oracles obtained from formal specification of the system behaviour. For instance, Frank and Doong developed the ASTOOT tool which generates test suites along with test oracles from algebraic specifications (Doong and Frankl 1994). In their work, test oracles can be generated by the ASTOOT tool and then used to verify the equivalence between two different executions scenarios. Specified oracles are effective in finding system failures but their success depends heavily on the availability of formal specification of the system behaviour. However, the vast majority of systems lack an accurate, complete and up-to-date machine readable specification. Therefore, the applicability of specified oracles is limited.

Implicit oracles are test oracles generated without requiring any domain knowledge or formal specification to implement. Hence, they can be applied to all runnable programs in the general sense. For example, in the fuzzing approach proposed by Miller et al. (1990), the main principle is to generate random inputs and attack the system to find faults which cause the system to crash. If a crash is spotted, then the fuzz tester reports the crash with the set of inputs or input sequences caused it. The fuzzing approach is widely used in the security vulnerabilities detection area such as buffer overflows and memory leaks etc.

Derived oracles are synthesised from properties of the system under test, or several artefacts other than the specification (e.g. documentation and system execution information), or other versions of the system under test. For instance, metamorphic testing has been used to test search engines such as Google and Yahoo (Zhou et al. 2012), and the BERT

[3] An interesting question arises if no failing outputs are found in the small clusters: should the developer stop checking or continue to explore clusters of increasing size? In our studies so far this has never happened, but our ultimate goal is to develop this technique to the extent where we could be confident that any failures are very likely to appear in small clusters.

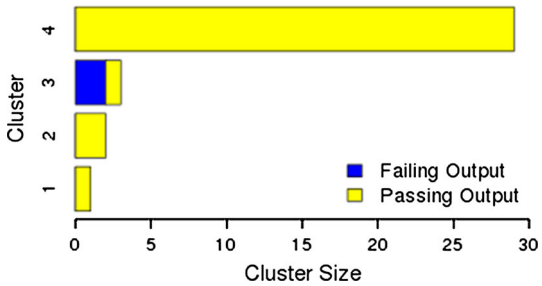


Fig. 2 Clusters generated from triangle example data. The clustering technique groups the test inputs and outputs into four clusters. There are two failing tests which appear in cluster 3. The remaining 33 are all passing tests

tool may be used to identify behavioural differences between two versions of a program from examining inputs, outputs, return values and program states (Jin et al. 2010)—a promising regression testing approach but one which relies on the presence of a previous reference version of the software, which may not always be available or suitable.

Our work is rooted in the area of derived oracles from system executions; therefore, the related work can be divided in two main sections: test oracles based on invariant detection and test oracles based on anomaly detection.

3.1 Test oracles based on invariant detection

Program behaviours can be automatically checked against given invariants for violations. Therefore, invariants can be used as test oracles to find out the correct and incorrect output. Invariants are often inserted into the code by the developers, but this again can be a costly exercise and an additional burden at the time of coding. Daikon can be used to learn and infer invariants from program executions dynamically by using a collection of inputs (test cases), monitoring key values (class attributes, method entry and exit points, loop invariants etc.) and then making inferences from this large set of observations (Ernst et al. 2007). Sekar et al. (2001) proposed an approach to learn Finite State Automata (FSA) by using sequences of systems calls. Their approach deals with system security and is aimed at detecting anomalous sequences of system calls which are likely to point to intrusion attempts and malware. Hangal and Lam build up invariants over program variable from the executions of the passed tests and then use any violations of these invariants to identify potential bugs (using the DIDUCE tool) (Hangal and Lam 2002).

3.2 Test oracles based on anomaly detection techniques

Chandola et al. (2009) define anomaly detection as a matter of spotting patterns in data that correspond to abnormal behaviour. This concept is illustrated in Fig. 3—the unfilled circles represent regions of normal behaviour, whereas the filled points represent anomalous data. The aim of the work reported in this paper to investigate whether software bugs generate a non-conformant pattern of behaviour that can be distinguished from the conformant or normal behaviour—in other words, in Fig. 3 do the groups of unfilled points corresponded to passed tests and the filled ones with failures? If this is the case, then the possibility of detecting bugs automatically can be raised.

The main principle of creating test oracles in this context is to hypothesise a formal model of program behaviours from sets of observations. There is a large body of work on using anomaly detection strategies such as clustering and classification techniques to support software testing tasks. However, these typically operate on quite different types of

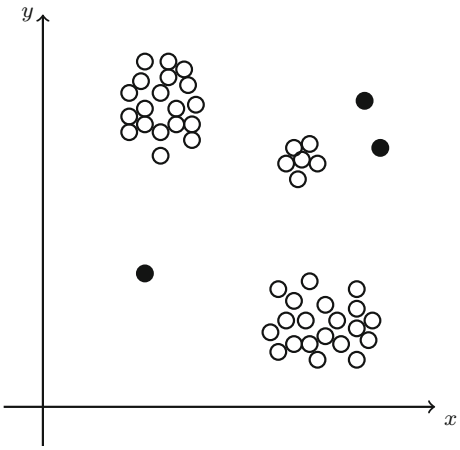


Fig. 3 Principle of anomaly detection. Non-anomalous items (*unfilled circles*) group together in larger clusters while anomalous ones (*filled circles*) are left isolated

data set (e.g. execution traces), or utilise semi-supervised or supervised learning strategies (such as the presence of a previous version of the program). Consequently, the application of anomaly detection strategies in this context has not been extensively investigated (for a detailed review of anomaly detection techniques and applications see the work of Chandola et al. (2009)). The following subsections discuss some recent work in this area. The work in those subsections will be classified in to three main categories: (1) unsupervised learning techniques; (2) semi-supervised learning techniques; (3) supervised learning techniques.

Unsupervised learning techniques do not require training data and thus are most widely applicable. The techniques in this category make the implicit assumption that normal instances are far more frequent than anomalies in the test data. If this assumption is not true, then such techniques suffer from a high false alarm rate. Examples of such work include that of Dickinson et al. (2001, 2001) who demonstrated the advantage of automated clustering of execution profiles over random selection for finding failures by using function caller/callee feature profiles as the basis for cluster formation. This work is in turn based on that of Podgurski et al. (1999), who used cluster analysis of profiles and stratified random sampling to calculate estimates of software reliability and found that failures were often isolated in small clusters based on unusual execution profiles. Our work is similar to this and explores the same observed hypothesis about the distribution of failures over clusters, but we investigate the use of test case input/output pairs (and input/output pairs combined with execution profiles) from the system under test instead of execution profiles alone.

Semi-supervised learning techniques typically assume that training data has labelled instances for only the normal class (i.e. a subset of passing test cases needs to be identified). A model is built for the class that corresponds to normal behaviour and then used to identify anomalies in the unlabelled test data. Podgurski et al. investigate how bugs could be classified when represented by a failed test that had the same cause (Podgurski et al. 2003). Their approach worked based on the analysis of the execution profile corresponding to reported failures of the test and was built on top of their earlier unsupervised learning system. Bowring and colleagues proposed an automatic classification of program behaviours using execution data which aimed at reverse engineering a more abstract description of system's behaviour (Bowring et al. 2004).

Supervised learning techniques assume the availability of a training data set which has labelled instances for normal as well as anomaly classes and is therefore the least generally

applicable. However, this has been successfully used in regression testing where a reference version of the software exists which makes accurate data labelling possible. For example, Vanmali et al. (2002) trained a multi-layer neural network on the original software application by using randomly generated test data that conformed to the specification. When new versions of the original application are created and regression testing was required, the tested code was executed on the test data to yield outputs that are compared with those of the neural network. Frounchi et al. explored the possibility of using supervised learning as test oracle (Briand 2008) within the image processing domain.

4 Clustering techniques

Clustering aims to partition a population of objects, each containing various attributes, into groups in such way that objects with similar values are placed in the same cluster, whereas those with dissimilar ones are placed in different clusters. The similarity of objects can be decided by using different distance metrics (discussed in more detail in Sect. 5.2). In this work the objects of interest are observations from program executions—test inputs and outputs and execution traces—and the aim of clustering is to separate the passing and failing executions. There is a very large variety of approaches towards clustering and so far this work has explored the use of the following algorithms: agglomerative hierarchical clustering, density based spatial clustering of application with noise clustering (DBSCAN) and expectation-maximization clustering (EM). The following subsections give a brief description of each approach. For further details on the techniques the reader is referred to the work of Han et al. (2012) or Witten and Frank (2005) for example.

4.1 Agglomerative hierarchical clustering

The agglomerative hierarchical algorithm is an example of a clustering approach that aims to build a hierarchy of objects. The core principle of this type of clustering method is that the objects are more related to nearby objects (as defined by the distance metric) than to objects farther away. A hierarchical clustering method can be either agglomerative or divisive, depending on whether the hierarchical decomposition is formed in a bottom-up (merging) or top-down (splitting) fashion.

Agglomerative hierarchical clustering initially assigns each object to its own cluster, calculates the distance between each pair of clusters, and combines the most similar ones. This process is repeated, building larger and larger clusters at higher levels of the hierarchy, until no close similarity or dissimilarity between two clusters can be found.

Divisive hierarchical clustering operates in the opposite fashion, initially assigning all objects into one cluster and then dividing this main cluster into smaller ones based on object dissimilarity until no further splits can be made.

In both approaches the user can specify the desired number of clusters as a termination condition.

4.2 DBSCAN

Density based spatial clustering of application with noise is an example of density based clustering approach, grouping together those objects that are close neighbours which allows it to find arbitrarily shaped clusters. Unlike agglomerative hierarchical clustering,

the number of clusters can be determined automatically after specifying two key parameters: the minimum number of points in a cluster and the distance between them. The approach also supports the notion of an outlier—objects not belonging to any cluster. A cluster is defined as containing at least a minimum number of points (MinPts), every pair of points of which either lies within a user specified distance (ϵ) of each other or is connected by a series of points that each lie within distance ϵ of the next point in the chain. Smaller values of ϵ yield denser clusters. Based on the value of ϵ and the minimum cluster size, it is possible that some objects will not belong to any cluster (these outliers are considered as noise).

4.3 Expectation-maximization (EM) clustering algorithm

The *EM* clustering algorithm is an example of probability based clustering approach. In contrast to an approach such as *k-means* clustering, in which a fixed number of clusters (k) is given at the outset and objects are assigned to those clusters so that the means across clusters (for all objects) are as different from each other as possible, *EM* works purely from the set of objects without any a priori information to find the most likely set of clusters from a probabilistic perspective. *EM* operates iteratively to assign data objects to clusters and update the parameters of the probability distributions governing the various clusters until the best model is found.

5 Experimental evaluation

Two main experiments were run to evaluate the effectiveness of clustering techniques in separate failing and passing tests.

Experiment 1: In the first experiment the input to the clustering algorithms consisted of just the test case inputs along with their associated outputs.

Experiment 2: The second experiment extended this by adding to the input/output pairs their corresponding execution trace.

The main hypothesis under investigation being: "Normal data instances belong to large and dense clusters, while anomalies (failures) either belong to small or sparse clusters". In other words, is the execution data which falls outside the clusters or appears in small *(sparse)* clusters indicative of bugs? Data about the distribution of failures over clusters, the impact of the number of clusters, the density of clusters, and the number of faults revealed per cluster were analysed to examine this hypothesis. This section describes the framework used for the experiments.

5.1 Subject programs

Versions of three subject programs were used in this study: the NanoXML XML parser system, the scalable internet event notification architecture system (Siena), and the Sed stream editor. All are available from the Software Infrastructure Repository (SIR),[4] are non-trivial systems, have several versions with well-documented faults embedded within them (these are either real or seeded—typically coming from faults in previous versions of

[4] http://sir.unl.edu/portal/index.php.

the system), and also come with test suites—an important factor as having sets of good, but independently created, tests is vital for this experiment.

NanoXML NanoXML is a non-GUI based XML parser written in Java. NanoXML consists of a component library and an application JXML2SQL which takes as input an XML file and either transforms it into a html file, showing the contents in tabular form, or into an SQL file. NanoXML has 24 classes, five versions (although the fourth version was excluded as it contains no faults), each containing multiple faults—seven in each of versions 1–3 and eight in version 5—and 214 test cases. The error rates in all faulty versions ranged from 31 to 39 % (the error rate is the proportion of the supplied test cases which will fail due to the seeded faults).

Siena Scalable internet event notification architecture (Siena) is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks. Siena is responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering those notifications to the clients via access points. Siena contains 26 classes (nine in its core and 17 which constitute an application), 567 test cases and seven faulty versions: three with single, and four with multiple ones. Versions with multiple faults (V1,V3,V5 and V7) have been excluded from this experiment for the time being because of the absence of a fault matrix (a simple way of establishing which test cases are responsible for revealing which fault). Therefore, only V2, V4 and V6 are included in the experiment, each having a single fault and an error rate of 17 %.

Sed Stream editor (Sed) is a Unix utility that parses and transforms text using a simple compact programming language. Sed takes a set of commands and a text stream and performs some operation (or set of operations) on the input stream. Sed is typically used for extracting part of a file using pattern matching or substituting multiple occurrences of a string within a file. Sed is written in C, has 225 functions, 370 test cases and seven versions with multiple faults. Only one version was used in this experiment: version 5 which contains 4 faults and has an error rate of 18 %.

5.2 Experimental set-up

The main components of the experiment were: a set of programs with known failures, a set of test case inputs for each program, a way to determine whether an execution of each test was successful or not (passed or failed), and a mechanism for recording the execution trace taken through the program by each test. The seeded versions of the subject programs were run on the test cases to produce the associated outputs, and Daikon (Ernst et al. 2007) was used to obtain the execution traces. The resulting set of test case input/output pairs was augmented with their associated execution traces, transformed to reduce the volume of data (traces are often very large), and then analysed using several clustering algorithms. Knowing which data objects corresponding to failed test cases enabled us to determine how well the clustering algorithms performed. Each of these steps is described in more detail below.

Test case input/output pair collection The subject programs come with Test Specification Language (TSL) test suites and tools to run these automatically (details are available from the SIR repository and the article by Do et al. (2005)). Test cases which failed to produce any output were discarded (seven out of 214 for NanoXML, and 73 out of 567 for Siena, and seven out of 370 for Sed, giving final test case numbers of 207, 494 and 363).

Execution trace collection Daikon was used to instrument the subject programs in order to collect the execution traces used in the second experiment. For the subject programs, we

execute each test case to produce its associated execution trace. Daikon allows programs to be monitored and traced at varying levels of granularity, but for this study we extracted sequences of method invocations (entry points) and method exits in the order they occurred during test execution.

Identification of failures The NanoXML and Sed systems come with matrices which map test cases to failures corresponding to faults and makes the identification of faults effectively automatic. Siena has no such fault matrix so the test outputs of the original version were compared with that of the faulty ones to find the failing tests.

Data transformation To be acceptable to the various clustering algorithms, the data requires processing before it can be analysed. The processing procedures differ from one data type to another—for instance numeric data sometimes requires normalisation. All systems used in the experiment work with textual input and produce textual output. Very often there is little semantic information in such data and a lot of noise, so to minimise the content (and redundancy) but still retain any uniqueness, the data (test case input/output pairs) were transformed by a simple process of tokenisation. The tokenisation method is widely used in the area of text mining to produce a suitable set of attribute vectors to build a classification model (a problem not dissimilar to the one we are dealing with) and is also suggested by Witten and Frank (2005). Several transformation methods such as hash coding, Huffman coding strategies and others were examined, but tokenisation turned out to be the most suitable one. Table 2 shows an example of this for NanoXML and Siena. Notice that the parameters for Siena commands were all encoded as "1" as they remained unchanged between input and output.

The Sed test data (input/output pairs) consists of a command line which contains 2 main parts: the parameters identifying the operations to be performed and a text file that needs to be modified which therefore forms both part of the input and output. Therefore, the data were transformed in a slightly different way compared to NanoXML and Siena: all input components remained unchanged except the filename (e.g. "../inputs/default.in") which was encoded as the token "<1>" as the file itself contains only the text to be modified. Trying to tokenise the file to be modified (and its modified version) failed to reduce the size of the output sufficiently and so for output part the diff utility (a data comparison tool) was

Table 2 Example coding of input/output pairs

	Input	Output
Nanoxml	**Flower colour="Red"** **Smell="Sweet"** **Name="Rose"** **Season="Spring"**	Xml element name is: **Flower**
Encoding	**FCRSSNRSS**	**F**
Siena	Filter senp{x=0}**filter**{**x=20 y=30 z=10**}Event senp{x=0}**event**{**x=20**}senp{x=0}**event**{**y=30 z=10**}	**Subscribing** for **filter**{**x=20 y=30 z=10**}**publishing** for **event**{**x=20**}**publishing** for **event**{**y=30 z=10**}
Encoding	**F111E1E11**	**SF111PE1PE11**
Sed	**sed -e 's/dog/cat/' ../inputs/default.in**	The modified text file (change and add operations)
Encoding	**sed-es/dog/cat/<1>**	**114a36c34c29c26c3|4c0a**

used to calculate the differences between the input text file and its modified form (this process reports how to change the first file to make it match the second file with specific operation that needs to be performed such as "a" for add and "c" for change). The magnitude of the compression achieved by this method is hard to quantify, depending as it does on the file and the modifications, but it typically yielded a much smaller representation of the output data. Table 2 shows an example of this coding strategy.

As explained earlier, each test input/output pair was augmented with its associated execution trace. Such traces are often very long (hundreds, and in some cases thousands of entries), and each entry in a sequence is often a full Java method signature including package name, class name, method name, and parameters (along with their respective long signatures). This required more compression than could be provided by simple tokenisation so the trace compression algorithm developed by Nguyen et al. (2013) was used. The algorithm replaces the collections of method sequence entry and exit values with their hash keys, consisting usually of just 1 or 2 characters. It takes into account the occurrence frequency to assign shorter hash keys for entries that are most frequent. Table 3 shows a sample of sequences for one of collected traces and their hash key values (for space reasons, just 3 sequences are included rather than all sequences of that trace) which are then concatenated to produce one single string. The obtained trace from the example in this table is **0LA37**...

Finally all the data items can be combined into vectors that forms the input to the clustering algorithm. These vectors are built from two components for the first experiment (test input and output) and three for the second experiment (test input, output and execution trace). So if the NanoXML example from Table 2 above generated the trace fragment shown in Table 3, then the vectors would take the form of **<FCRSSNRSS, F>** for experiment 1 and **<FCRSSNRSS, F, 0LA37...>** for experiment 2. This structure is repeated for all the input/output [trace] combinations for each test case.

Perform clustering Agglomerative hierarchical clustering was used in experiment 1. The second experiment extended this to include also DBSCAN and EM clustering. Agglomerative hierarchical clustering has been used by other researchers for some similar types of problem and shown to perform reasonably well [e.g. Dickinson et al. (2001), Dickinson et al. (2001), Yan et al. (2010), Yoo et al. (2009)] and is also recommended by Witten and Frank (2005) as the most suitable solution for nominal and string data (which the coding systems produce for two subject programs). In contrast, DBSCAN and EM were chosen because of their ability to determine the number of clusters automatically rather

Table 3 Example coding of sequence traces

Sequence traces	Hash key values
net.n3.nanoxml.XMLElement. getFullName():::EXIT283	0L
net.n3.nanoxml.XMLUtil.skipWhitespace (net.n3.nanoxml.IXMLReader,char, java.lang.StringBuffer, boolean[]):::ENTER	A
net.n3.nanoxml.StdXMLReader. getEncoding(java.lang.String):::ENTER	37
⋮	⋮

than have to specify them at the outset (one of the limitations observed in the first experiment).

A range of distance measures were initially explored such as Euclidean distance, Minkowski distance, Manhattan distance and edit distance in order to establish the most suitable measure for the experiments proper. The first three were similar in terms of the performance and principle. However, edit distance did not perform well and agglomerative hierarchical clustering consistently assigned all input/output pairs into one cluster even when the clusters count was increased. After exploring these various alternatives, Euclidean distance was settled on as the measure of (dis)similarity between two objects. The WEKA toolkit[5] used in this study computes this by converting all nominal attributes into binary numeric attributes. So, an attribute with k values is transformed into k binary attributes (using the one-attribute-per-value approach) (Witten and Frank 2005). Thus, all attributes values are binary: being either a numeric attribute or a synthetic binary attribute that is treated as numeric. The squared Euclidean distance sums the squared differences between these attributes: a zero sum indicates agreement (similarity), but a non-zero sum suggests a dissimilarity.

The consequence of choosing Euclidean distance is that nominal or categorical data (such as the inputs, outputs and traces used in these experiments) are only considered equal if they are identical. Any form of difference, no matter how small or large, causes them to be considered unequal. This means that two traces may differ in just one method call out of thousands but are considered as different as two that had no method calls in common. This might seem an odd decision but the rationale behind this is that even a slight difference in an execution trace may be indicative of an error. Using other measures would mean such a difference was hardly perceptible and could easily be missed. The impact of this decision, along with other distance measures, is something that needs to be explored further in the future.

In addition to a similarity metric, agglomerative hierarchical clustering requires a linkage metric which is used to determine when clusters should be merged or split. There are three approaches: *Single Linkage* calculates the minimum distance between an object in one cluster and an object in another, *Average Linkage* computes the mean distance between objects in the two clusters, and *Complete Linkage* is based on the maximum distance between objects. All three are explored in this study.

Number of clusters For agglomerative hierarchical clustering, the number of clusters needs to be provided as parameter. This can clearly have a significant impact: too many clusters results in fragmentation and too few in over-generalisation. Therefore, a number of different cluster counts were explored based on a percentage of the number of subject program test cases: 1, 5, 10, 15, 20 and 25 %.

The number of clusters for EM is determined automatically by *cross validation*, a technique often used in classification (Witten and Frank 2005). A given data set is firstly divided into m parts. Next, $m-1$ parts are used to build a clustering model, and the remaining part used to test the quality of the clustering. This process is repeated m times to derive clusterings of k clusters by using each part in turn as the test set. The average of the quality measure is taken as the overall quality measure. Then, the overall quality measure with respect to different values of k is compared to find the best number of clusters that fits the data.

The DBSCAN algorithm uses two specified parameters (ϵ: the radius parameter, and MinPts: the neighbourhood density threshold—see Sect. 4.2) to determine the number of

[5] http://www.cs.waikato.ac.nz/ml/weka/.

clusters automatically. For our experiments, we found that the parameters which gave the best results were $\epsilon = 1.5$ and MinPts = 1.

Small cluster size One of the key elements of this research is the hypothesis that failures tend to congregate in small clusters. But what is a small cluster? For these initial studies, small is defined as less than or equal to the mean of the cluster size (the remainder being considered as large). For the purposes of this experiment, all clusters were examined to determine the proportion of failures contained therein, but in practice is it envisaged that only small clusters would be inspected and larger ones ignored. The definition of 'small' and 'large' is quite coarse in this instance. One of the topics of future research is to more accurately define what can be considered to be small clusters.

5.3 Evaluation of clustering techniques

The performance of the clustering algorithms can be assessed by looking at the way that failures are distributed over the small clusters (the definition of "small" is flexible so what follows is a general definition). To capture more accurately for this experiment, we used the *F-measure*—a combination measure of *Precision* and *Recall* (widely used measures in information science domain). These measures in turn rely on the concepts of true positives (TP), false positives (FP) and false negatives (FN) which are defined in this context as follows:

TP: A failing test result that appears in a small cluster.
FP: A passing test result that appears in a small cluster.
FN: A failing test result that appears in a large (i.e. not small) cluster.

Precision is defined as the ratio of "correctly clustered" failures (i.e. failures that appear in small clusters) to the sum of all the entries in the small clusters:

$$\text{Precision (PR)} = \frac{(\text{TP})}{(\text{TP} + \text{FP})} \tag{1}$$

Recall is the ratio of "correctly clustered" failures to the total number of true failures (failures appearing in both small and large clusters):

$$\text{Recall (RE)} = \frac{(\text{TP})}{(\text{TP} + \text{FN})} \tag{2}$$

The F-measure—the harmonic mean of precision and recall—combines these two as follows:

$$\text{F} - \text{measure} = 2\frac{(\text{PR} \times \text{RE})}{(\text{PR} + \text{RE})} \tag{3}$$

In this study, we have defined small clusters as those being of average size or less (i.e. the total number of passing and failing outputs divided by the number of clusters). Further work in this area will explore other values of small.

To illustrate the process of the evaluation, we introduce a small example which shows how the small cluster size, precision, recall and F-measure are computed. Assume that a system under test generates 21 data points during execution of its set of test cases. The system contains three faults (referred to as F1, F2 and F3) which cause failures which appear in the output 4, 4 and two times respectively. The remaining 11 test outputs were all passes (we do not need to distinguish amongst these). Again assume that after applying

clustering, six clusters were created which grouped the outputs as follows: *(f1, f2, f3, p, p, p), (p, p, p, p, p), (f1, f2, p, p), (f1, f2, p), (f2, f3), (f1)*, where *fn* corresponds to a failure associated with fault *n* and *p* corresponds to pass execution. This can be illustrated graphically as shown in Fig. 4 (where the clusters are sorted in increasing order of size on the y-axis and the "cluster count" legend is just an arbitrary value allocated to a cluster). This representation allows us to see the distribution of failures over the clusters.

The key values are computed as follows:

- Small clusters are those of average size or less (i.e. (number of data points)/(number of clusters)). In the above example, the average cluster size is (21/6) = 3.5, so the small clusters are all of these containing $\leq$ 3 data points (i.e. clusters 1, 2 and 3).
- Precision: Five of the outputs in the 3 small clusters are failures (TPs) and one is a pass (FP), so PR = 5/(5 + 1) = 0.83
- Recall: Five of the outputs in the 3 small clusters are failures (TPs) but 5 failures also ended up being allocated to the "large" clusters (TNs), so RE = 5/(5 + 5) = 0.5
- The F-measure is then $2 \times (0.83 \times 0.5)/(0.83 + 0.5) = 0.62$.

6 Experiment 1 (clustering test input/output pairs): results and discussion

This first experiment explored the use of clustering to group data composed just of test case inputs and their associated outputs.

6.1 Distribution of failures

The first question to explore is whether failures are distributed in a random pattern or whether they tend to congregate in the smaller clusters as hypothesised. Figures 5, 6, 7, 8, 9 and 10 show bar charts representing the cluster size and composition for all versions of NanoXML, Siena (faulty version 2), and Sed using agglomerative hierarchical clustering with average linkage. The results are interesting and in several cases (NanoXML versions 2 and 3 and Siena version 2) it can be seen from these that failures in the test input/output pair population tend to cluster together and these clusters tend to be the smaller ones. This effect is less pronounced in NanoXML versions 1 and 5 where the smallest clusters also tend to contain more of the passing cases. The pattern for Sed is quite different—there are a very large number of small clusters rather than a gradually increasing distribution as in the other cases, and these contains a mixture of both passing and failing cases. Overall there is some support for the main hypothesis behind this work, that failure tends to gravitate

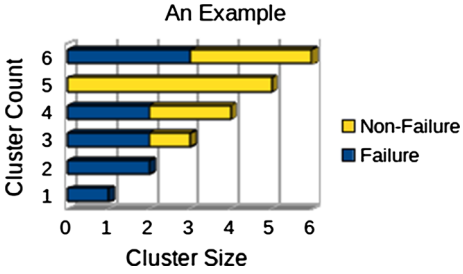


Fig. 4 Evaluation example

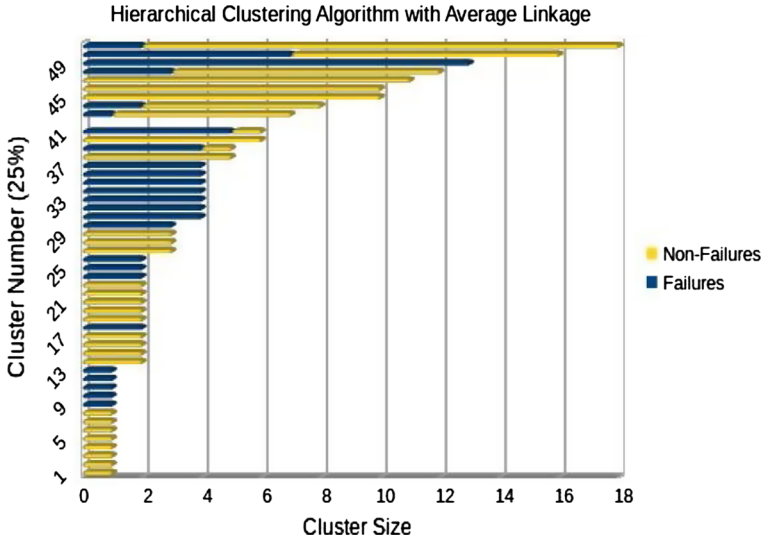


Fig. 5 Hierarchical clustering algorithm with average linkage for NanoXML (version 1)

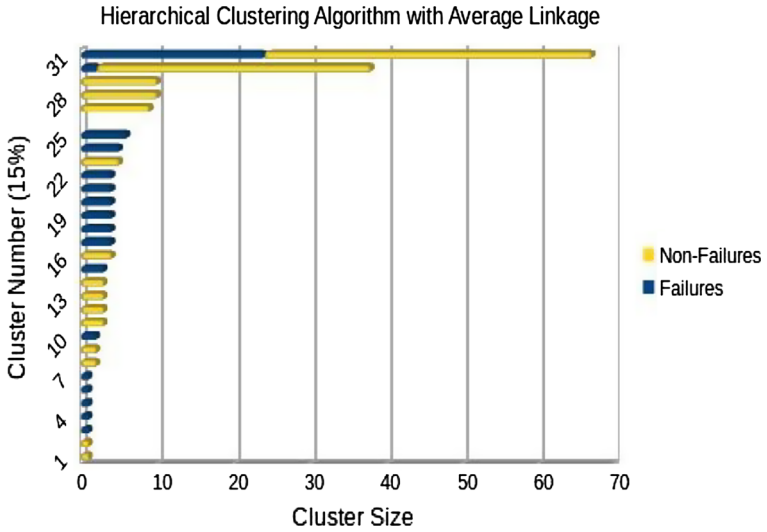


Fig. 6 Hierarchical clustering algorithm with average linkage for NanoXML (version 2)

towards the smaller clusters but it is by no means universal. The following sections examine this in more detail.

6.2 Failures found versus cluster counts and cluster sizes

To investigate this observation further, we examined the population of input/output pairs that were in small clusters (defined as being of average size or less) and corresponded

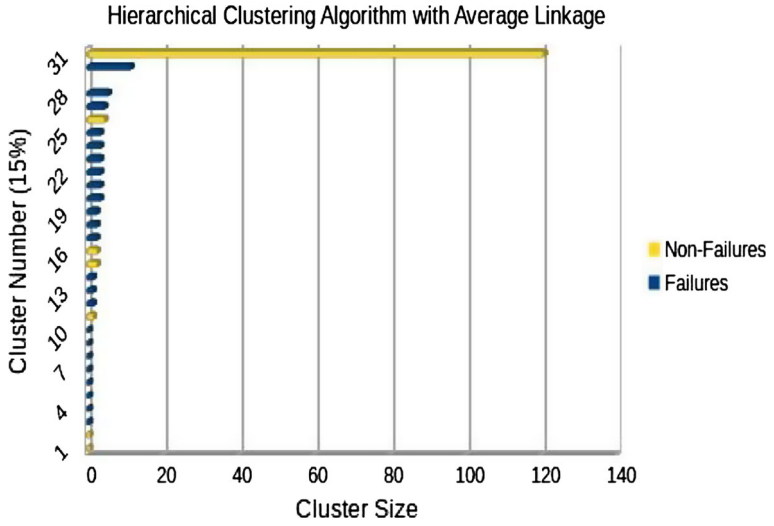


Fig. 7 Hierarchical clustering algorithm with average linkage for NanoXML (version 3)

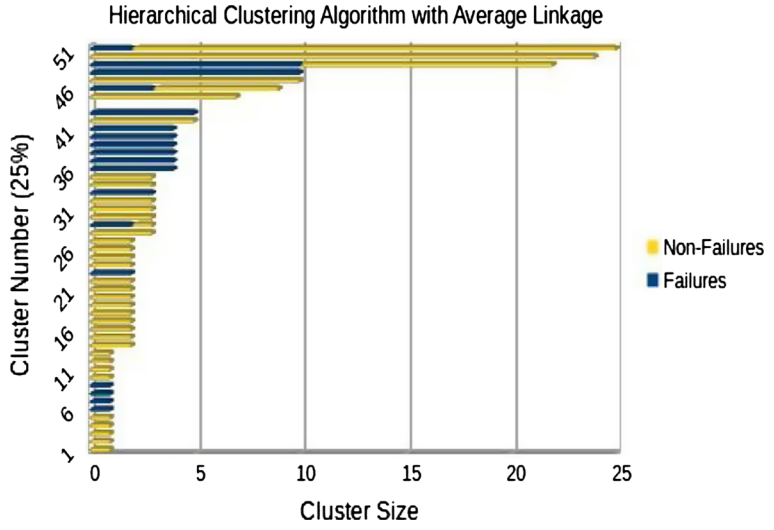


Fig. 8 Hierarchical clustering algorithm with average linkage for NanoXML (version 5)

to failures. Tables 4 and 5 show, for varying numbers and sizes of clusters over all systems and for the three different linkage metrics that may be used with agglomerative hierarchical clustering (Average, Single and Complete), the percentage of all data points corresponding to failures. The first column (Cluster Count %) defines the number of clusters the algorithm is charged with creating expressed as a percentage of the number of test cases. So, for NanoXML a value of 10 in the Cluster Count % corresponds to 21 as it has 207 tests, for Siena this would be 50 as it has 494 test cases, and for Sed which has 370 tests it would be 37. The second column (Cluster Size %) is the average size of the clusters that are created by the algorithms, again expressed in terms of the number of tests. So as

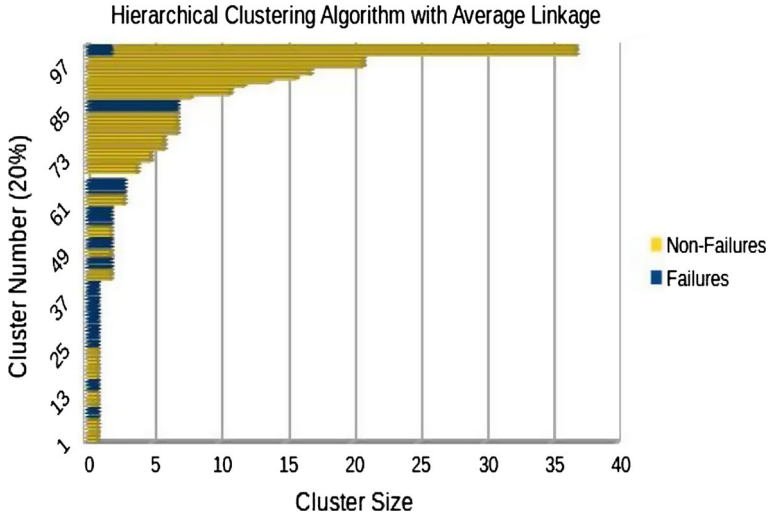


Fig. 9 Hierarchical clustering algorithm with average linkage for Siena (version 2)

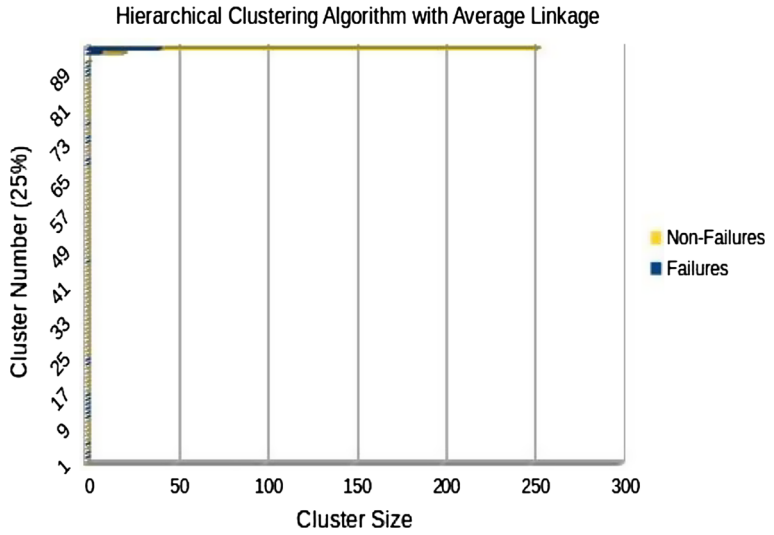


Fig. 10 Hierarchical clustering algorithm with average linkage for Sed (version 5)

the values in Cluster Count % column increase, so do the number of clusters created which leads to a corresponding decrease in the average size of the clusters. The subsequent columns refer to the version number of the program. Note that the faults in Siena changed the same output data in all versions, even though they are distinct faults, so only the results from one version are considered since there is nothing to be gained from examining the other versions.

Considering the results for NanoXML (Table 4), the data shows that when the cluster counts are between 15 and 25 % of the number of test cases (corresponding to cluster sizes of around 3 % of the number of test cases—i.e. around six data points for NanoXML), well

Table 4 Composition of small clusters in terms of failures and F-measure versus cluster size for hierarchical clustering with different linkage metrics for NanoXML

Cluster details		NanoXML version							
(% tests)		V1		V2		V3		V5	
Count (%)	Size (%)	%	F	%	F	%	F	%	F
Single linkage									
1	50	0	0	0	0	0	0	0	0
5	10	14.81	0.2398	18.30	0.2763	7.14	0.1187	26.15	0.1367
10	3.5	53.08	0.6322	63.38	0.7142	50.0	0.5931	40.0	0.4521
15	3	56.79	0.6215	63.38	0.6521	65.71	0.7105	61.53	0.6011
20	2.5	56.79	0.5679	63.38	0.6080	72.85	0.7554	66.15	0.5771
25	2	56.79	0.5379	63.38	0.5555	74.28	0.6886	66.15	0.5407
Average linkage									
1	50	7.4	0.070	28.1	0.0394	100	1	10.76	0.0907
5	10	56.79	0.6012	63.38	0.6337	34.28	0.4658	26.75	0.2984
10	6.25	56.79	0.5840	63.38	0.6337	45.71	0.5764	61.53	0.5969
15	3.25	56.79	0.5643	63.38	0.6293	82.85	0.811	52.30	0.5311
20	2.5	51.85	0.5121	54.92	0.5164	75.71	0.7065	52.30	0.4329
25	2.25	65.43	0.5578	61.97	0.5534	75.71	0.6623	61.53	0.4847
Complete linkage									
1	50	12.30	0.1209	28.1	0.0436	100	1	10.76	0.0887
5	10	12.30	0.1673	29.57	0.2673	20.0	0.3333	26.15	0.2832
10	6.25	35.80	0.3693	17.71	0.493	67.14	0.8033	44.61	0.3999
15	3.12	59.25	0.5643	46.47	0.4176	84.28	0.8193	55.38	0.4443
20	2.5	51.85	0.5123	54.92	0.5130	75.71	0.6541	52.30	0.4329
25	2.25	54.38	0.5145	64.78	0.5677	075.71	0.6623	53.84	0.4373

over 55 % of the data points are failures irrespective of which linkage metric is used, and over 60 % when the average linkage metric is employed. For Siena (Table 5) a similar pattern emerges but the best results are at the higher cluster count levels (20–25 %, possibly due to the larger number of test cases which gives an average cluster size of around 4) and tend to be over 70 %. The results for Sed (Table 5) are less dramatic and although a similar trend is displayed the failure density never reaches 50 %, peaking at just over 40 % when the complete linkage metric is used with an average cluster size of about 3. From the graphs shown earlier (Fig. 10), it was observed that Sed contained a very large number of small clusters and only one large cluster, rather than a steadily increasing cluster size which suggests that the data is very fragmented and the algorithm is clearly struggling to form larger groups of data items.

Even with the results from Sed, the findings lend support to the main hypothesis of this paper: As the number of clusters increases and their average size decreases, so the failure density of the small (less than average) sized clusters tends to increase. One case where this is not quite true is version 3 of NanoXML where the largest clusters contained the most failures: the input-output pairs corresponding to failures are so distinct from the rest that they were all grouped into one cluster (an impressive but probably unusual case!).

Table 5 Composition of small clusters in terms of failures and F-measure versus cluster size for hierarchical clustering with different linkage metrics for Siena and Sed

Cluster details		Siena version		Cluster details		Sed version	
(% tests)		V2		(% tests)		V5	
Count (%)	Size (%)	%	F	Count (%)	Size (%)	%	F
Single linkage							
1	19.8	0	0	1	19.8	14.9	0.266
5	4	3.57	0.0357	5	6.4	23.6	0.2954
10	2	40.47	0.3415	10	2.685	23.6	0.2903
15	1.21	48.80	0.4431	15	1.69	23.6	0.2427
20	0.79	72.61	0.6522	20	1.22	27.7	0.2375
25	0.6	60.71	0.5397	25	1	36.1	0.2723
Average linkage							
1	19.8	0	0	1	19.8	0	0
5	4	16.66	0.1325	5	6.46	9.7	0.1605
10	2	41.66	0.3909	10	2.628	12.5	0.1680
15	1.21	41.66	0.3703	15	1.563	18.0	0.2041
20	79	67.85	0.6194	20	1.08	25.0	0.2448
25	6	75.0	0.6236	25	0.808	29.1	0.2542
Complete linkage							
1	19.80	0	0	1	20	9.7	0.1662
5	4	33.33	0.1811	5	6.466	22.2	0.2641
10	2	47.61	0.3477	10	2.71	33.3	0.3836
15	1.21	66.66	0.4932	15	1.69	29.1	0.2995
20	79	72.61	0.6522	20	1.24	36.1	0.3074
25	6	67.1	0.5367	25	0.968	41.6	0.3058

Table 6 Percentage of failures and F-measure for EM algorithm

Systems	Cluster details		EM	
	Count (%)	Size (%)	%	F
Nanoxml V1	1.94	25	49.38	0.6557
Nanoxml V2	2.42	20.2	50.70	0.2950
Nanoxml V3	2.42	20	62.85	0.4398
Nanoxml V5	1.45	33	64.28	0.7757
Siena V2	2.02	16.66	35.71	0.2238
Sed V5	2.71	9.9	5.55	0.0655

Tables 6 and 7 show the results of clustering test inputs and outputs using the Expectation Maximisation and DBSCAN algorithms, respectively. Unlike agglomerative hierarchical clustering, neither of these algorithms require the number of clusters to be specified in advance. The results show that EM performs well with all versions of NanoXML but less so with Siena and very poorly with Sed. Interestingly, for NanoXML the number of clusters created is close to the best number when specified for agglomerative

hierarchical clustering. The results for DBSCAN are weaker for NanoXML and very poor for Siena but extremely encouraging for Sed, generating both a very high failure density in the smallest clusters and a reasonable F score. In the case of Sed DBSCAN has generated a very large number of small clusters (matching the pattern observed earlier in Fig. 10)—almost twice the number that was explored using agglomerative hierarchical clustering, which confirms our earlier observations about the data being very fragmented.

Although general pattern is for failure intensity to increase as the cluster size decreases, a trend which can also be observed in Figs. 11, 12 and 13 which present the percentage of failures found in the small clusters with different cluster counts in the subject programs (essentially a graphical summary of the data that appears in Tables 4 and 5), there are cases where the failure intensity peaks and then begins to drop (although not substantially) as the clusters are forced to fragment. An important lesson from this study is that the cluster size is crucial: too few and the technique may be ineffective but too many may cause the failure intensity to diminish as the clusters are forced to fragment. Identifying the ideal number of clusters (or similarly, the best parameters for algorithms such as DBSCAN) is something which needs further empirical investigation to establish.

Table 7 Percentage of failures and F-measure for DBSCAN algorithm (note for NanoXML epsilon = 0.9 Minpoints = 2, and for Siena and Sed epsilon = 1.5 Minpoints = 1)

Systems	Cluster details		DBSCAN	
	Count (%)	Size (%)	%	F
Nanoxml V1	19.9	2.68	25.92	0.2914
Nanoxml V2	19.9	2.68	22.53	0.2422
Nanoxml V3	19.9	2.70	25.11	0.2664
Nanoxml V5	19.9	2.60	16.92	0.1758
Siena V2	4.45	4.54	3.57	0.0357
Sed V5	48.23	1	83.3	0.3091

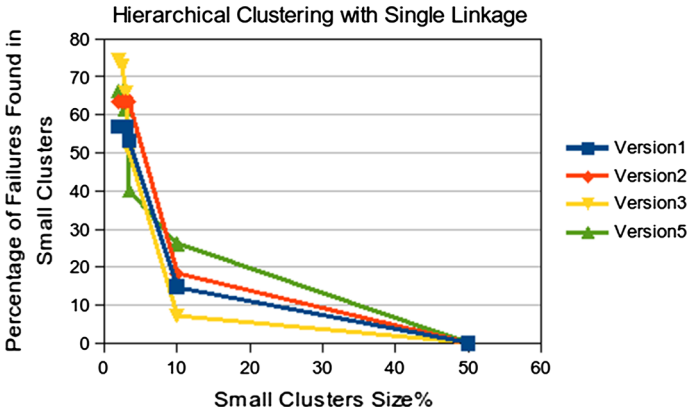


Fig. 11 Percentage of failures found over the smallest clusters for all Nanoxml versions using single linkage

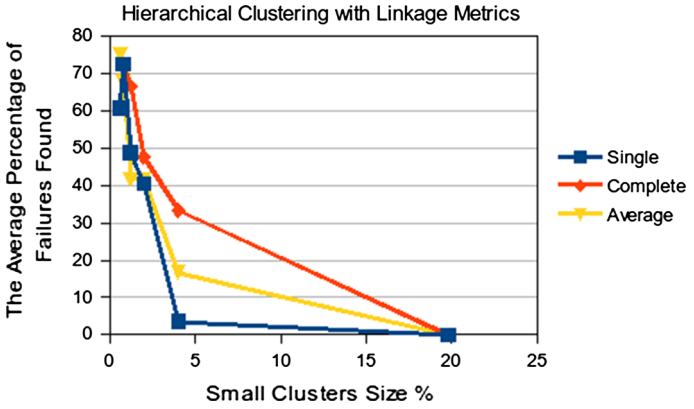


Fig. 12 The average percentage of failures found over the smallest clusters for all Siena versions using linkage metrics

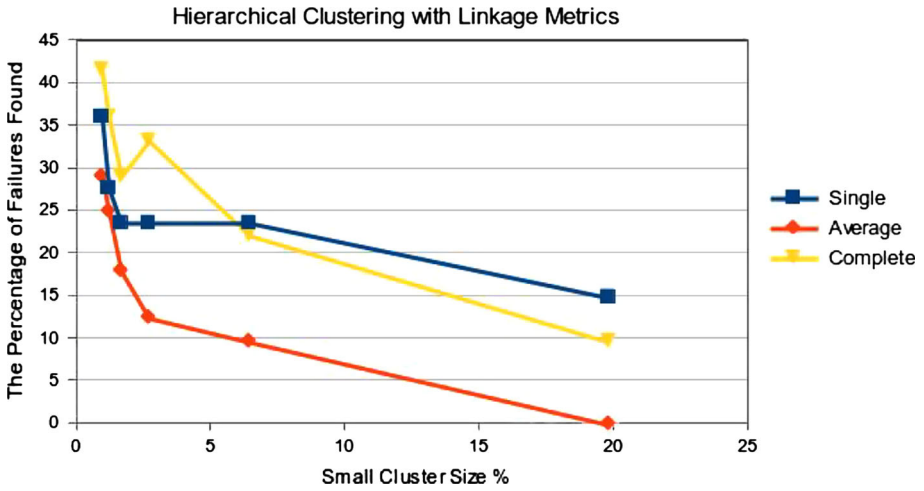


Fig. 13 The average percentage of failures found over the smallest clusters for Sed using linkage metrics

6.3 Failure density of smallest clusters

From the perspective of supporting the practising software engineer in their work and also in the construction of a test oracle, the interesting question concerns the return on investment: how many outputs need to be examined before a reasonable number of failures are observed? To answer this we examined in more detail the proportion of failing outputs appearing in the smallest sized clusters. The absence of a fault matrix for Siena makes this very time consuming to compute; therefore, only the results for the highest failure density clusters for NanoXML and Sed have been calculated so far. The results of this are summarised in Tables 8 and 9 and show the cluster size (the three values correspond to the absolute size of the cluster, the number of clusters of that size, and the size of the cluster and proportional to the test set size) and details of the failures found (the proportion, the actual failures indicated by 'Fn', and the number of occurrences of each failure). Failures

Table 8 Failure distribution over less than average-sized clusters for Nanoxml

Cluster size	Failures found	Cumulative
Version 1 (25 %)		
1, 13, 0.48 %	**F1**:3, **F2**:1, **F6**:1	3/7
2, 13, 0.97 %	F1:4, F2:2, F6:2	3/7
3, 4, 1.45 %	F6:3	3/7
4, 8, 1.94 %	F2:16, **F5**:8, **F7**:8	5/7
Version 2 (15 %)		
1, 7, 0.48 %	**F1**:3, **F2**:1, **F6**:1	3/7
2, 3, 0.97 %	F6:2	3/7
3, 5, 1.45 %	F6:3	3/7
4, 6, 1.94 %	F2:8, **F5**:8, **F7**:8	5/7
5, 2, 2.42 %	F2:5	5/7
6, 1, 2.91 %	F2:6	5/7
Version 3 (15 %)		
1, 10, 0.48 %	**F1**:4, **F2**:1, **F3**:1, **F4**:2, **F6**:1	5/7
2, 4, 0.97 %	F1:1, F4:2, F6:2	5/7
3, 5, 1.45 %	F4:6, F6:3	5/7
4, 6, 1.94 %	F2:8, **F5**:8, **F7**:8	7/7
5, 2, 2.42 %	F2:5	7/7
6, 1, 2.91 %	F2:6	7/7
Version 5 (25 %)		
1, 13, 0.48 %	**F1**:3, **F2**:1	2/8
2, 14, 0.97 %	F1:2, F2:2	2/8
3, 8, 1.45 %	F2:3	2/8
4, 7, 1.94 %	F2:28	2/8

associated with a new fault (i.e. not previously encountered) are indicated in bold font. The final column shows the cumulative count of unique faults observed (via their associated failures) over the total number of faults in the system. So, for instance, the first entry of Table 8 shows that for Version 1 using 25 % of the number of test cases to define the number of clusters, there were 13 clusters each of size 1 corresponding to 0.48 % of the number of test cases, containing failures 1 (three times), 2 and 6 (once each), giving a cumulative count of 3 out of a total of 7.

Table 8 shows that on average over all four versions a fair proportion of the failures—45 % (13/29)—are contained within the very smallest clusters (formed from just one or

Table 9 Failure distribution over less than average-sized clusters for Sed version 5

Complete linkage (25 %)		
Cluster size	Failures found	Cumulative
1, 62, 0.19 %	(**F1**:7, **F2**:4, **F3**:7)	3/4
2, 16, 0.38 %	(F1:2, F3:4)	3/4
3, 5, 0.57 %	(F2:3, F3:3)	3/4
4, 2, 0.76 %	(–)	3/4
5, 1, 0.96 %	(–)	3/4

two items). This is encouraging from a test oracle perspective: out of 43 outputs, 23 correspond to failures giving a failure density of 53 %. This initially good rate tails off until the cluster size reaches 4 and additional failures appear in the outputs (except for version 5). By this point an average of 66 % (19/29) of the failures have appeared in the clusters, albeit at the expense of having to examine more non-failing outputs and encountering duplicate failing outputs (but still giving a failure density of around 59 %). This failure density figure, combined with the fact that clusters tend to contain outputs associated with the same failure, means that in practice less than half of the outputs from a small cluster need to be checked before a failing output is encountered.

The results for Sed (Table 9) are less impressive but nevertheless encouraging. Even though the failure density is lower than for NanoXML, the failures are well represented in the smallest clusters: by examining these 3 out of the 4 failures would be encountered. On the downside the outputs of 62 small clusters (all of size 1) need to be checked, but this is still far less work than examining all 370 test outputs.

Of course, there are still additional failing outputs embedded in the larger clusters which cannot be ignored. This is clearly a weakness of the approach and one of the main topics of future work is to explore how these can be teased out into smaller clusters. A further feature of the clustering is that there is often number of independent clusters associated with the same failure (separated typically because the input/output pairs have different attribute values). This is also a challenge since finding the same failure appearing in several clusters can be quite frustrating for the individual charged with the task of checking outputs. Merging them together is not the answer as this will typically result in a larger cluster which may escape scrutiny, so some way of indicating similarity between them needs to be explored.

7 Experiment 2 (clustering test input/output pairs and execution traces): results and discussion

A second experiment was run to investigate whether collecting additional data in the form of the execution traces associated with each test case would improve the accuracy of the clustering performed in the first experiment by increasing in particular the failure density of the small clusters. Since this trace data can be quite extensive, it was compressed as described in Sect. 5.2. Apart from collecting and including this additional trace data in the clustering, all other aspects of this experiment were identical to the previous experiment.

7.1 Distribution of failures over clusters

Again, the first major question to explore is whether failures are distributed in a random pattern over the clusters or whether they gravitate towards the small clusters as hypothesised. To examine this a sample of the results are shown visually—space prohibits the inclusion of all the results, but the full set is available online.[6] Figures 14, 15, 16, 17, 18 and 19 show bar charts of the cluster composition for NanoXML (all faulty versions), Siena (just faulty version 2 as 4 and 6 produce an identical pattern as mentioned before) and Sed, where failing outputs are coloured blue and passing ones yellow. In these cases the cluster count for NanoXML is set at 15 % of the number of test cases (producing

[6] A complete sets of results can be found at: http://personal.strath.ac.uk/rafig.almaghairbe/.

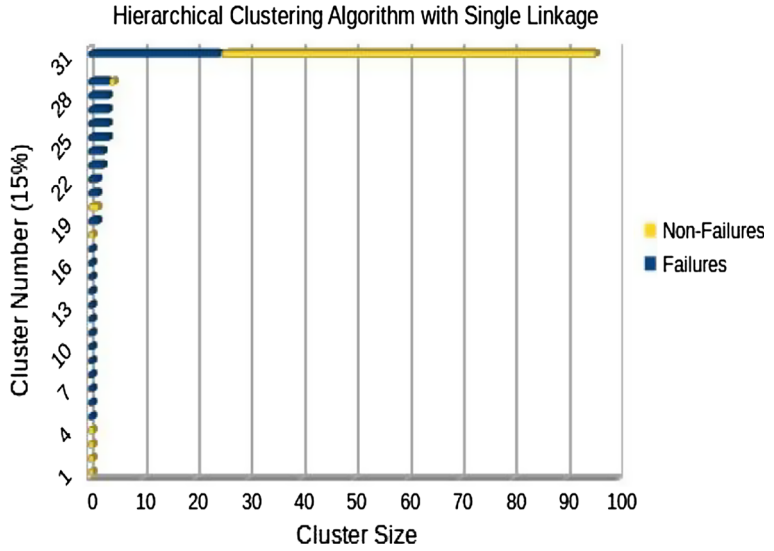


Fig. 14 Hierarchical clustering algorithm with single linkage for NanoXML (Version 1)

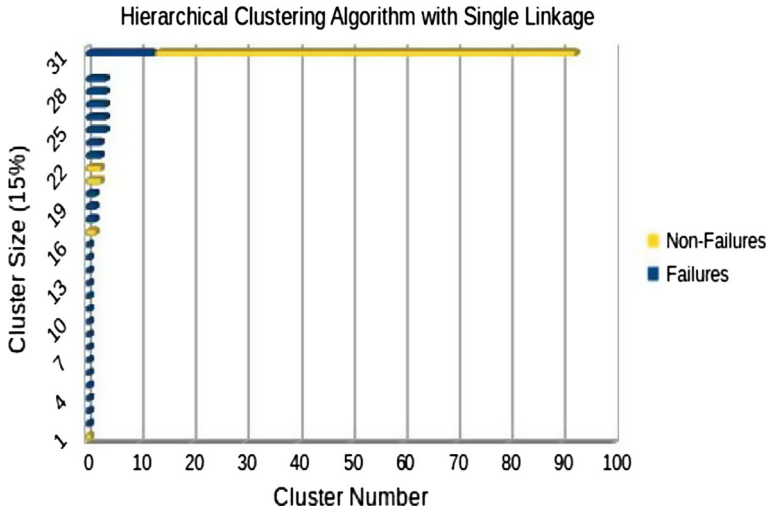


Fig. 15 Hierarchical clustering algorithm with single linkage for NanoXML (Version 2)

approximately 30 clusters), 20 % for Siena (producing just under 100 clusters) and 25 % for Sed (producing just over 90 clusters). In all cases the results are using agglomerative hierarchical clustering (DBSCAN and EM clustering algorithms were also used but tended to perform relatively poorly—something which is explored in more detail later).

It can be seen from these results that as in experiment 1 the failure data do tend to cluster together and these clusters are the smaller ones in most cases. There are some exceptions to this: for example for NanoXML version 5 the very smallest clusters are dominated by non-failing outputs, whereas the converse is true for the other versions, and

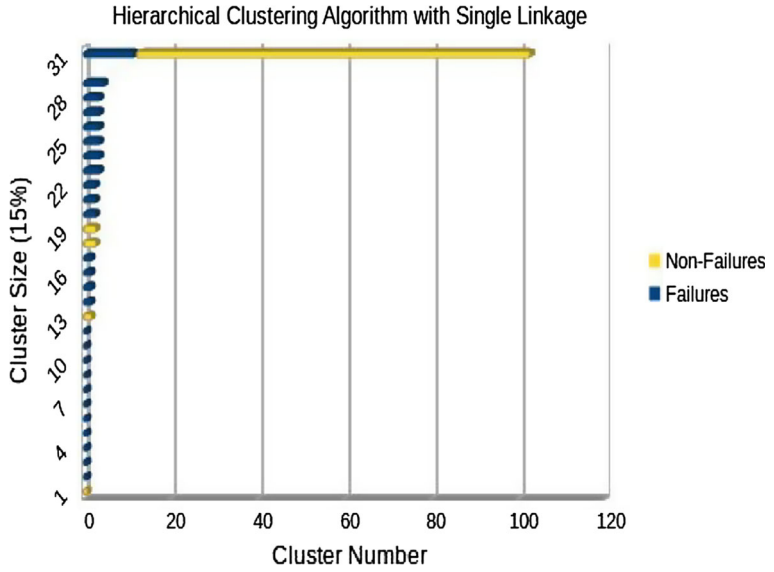


Fig. 16 Hierarchical clustering algorithm with single linkage for NanoXML (Version 3)

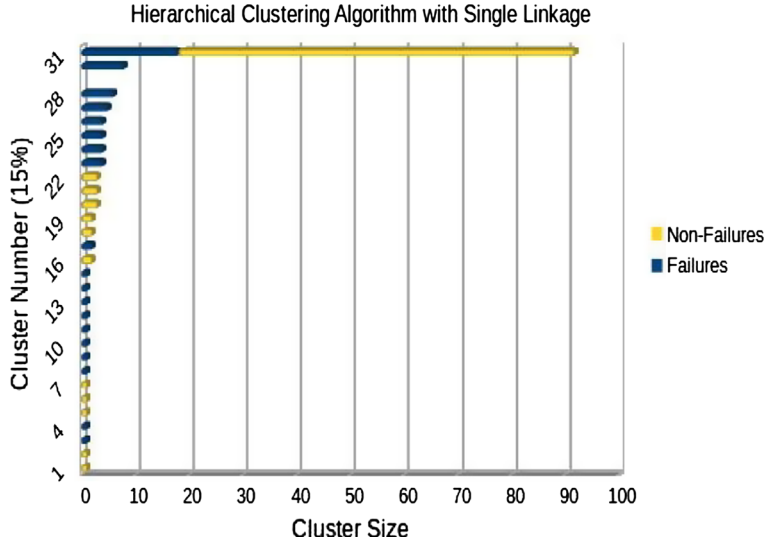


Fig. 17 Hierarchical clustering algorithm with single linkage for NanoXML (Version 5)

in all cases of NanoXML some failures creep into the largest clusters. The results for Siena are more consistent with a clear tendency for failures to gravitate towards the small clusters and away from the larger ones. The results for Sed are similar to experiment 1—many small clusters and one large cluster but this time with a few intermediate-sized ones. It must be stressed that these are selected, and very high-level, results (although others reflect a similar pattern) but it would seem that a substantial number of failures congregate in

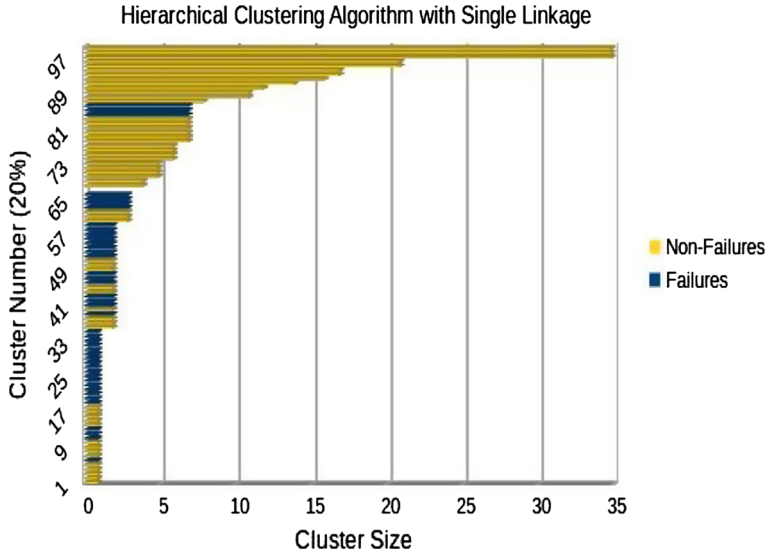


Fig. 18 Hierarchical clustering algorithm with single linkage for Siena (Version 2)

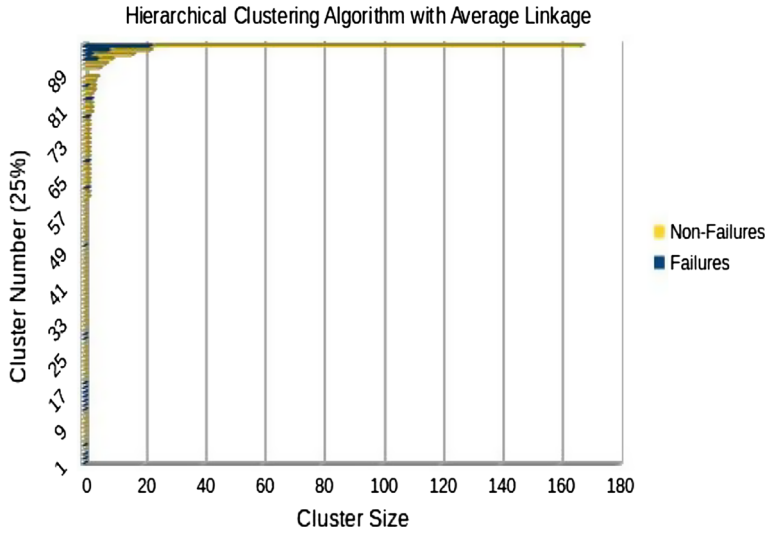


Fig. 19 Hierarchical clustering algorithm with average linkage for Sed (Version 5)

small clusters. The detailed composition of these small clusters is examined in more detail in the next section.

7.2 Failure composition of small clusters

This apparent observed tendency for failures to gravitate towards the smaller clusters need to be explored in more detail: the precise degree to which it occurs; the impact of the different clustering algorithms and parameters (especially the number of clusters); and

particularly the way that multiple failures are distributed (for example, in the case of several failures do they all appear in the small clusters or is one failure dominant?). To explore this principle further, we examined the population of the small clusters (defined as being of average size or less) for each of the algorithms, identified the percentage of these clusters that correspond to failures, and also used the F-measure to answer the point about the way that multiple failures are clustered.

Tables 10, 11 and 12 show, for NanoXML, Siena and Sed respectively, the results of applying agglomerative hierarchical clustering for different linkage metrics with varying numbers of clusters. The tables show the percentage of all data points in small (less than average sized) clusters that correspond to failures, and the F-measure for the small clusters. The percentage figure gives an indication of the failure density and the F-measure adds to this by considering the range of faults that are revealed by failures that appear in the small clusters (for NanoXML there are seven faults in versions 1–3 and eight in version 5). The first column (Count) defines the number of clusters the algorithm is charged with creating expressed as a percentage of the number of test cases. The second column (Size) is the average size of the clusters again in terms of the number of test cases. The cluster count figure has to be supplied as a parameter, whereas the size figure is a consequence of the number of clusters and is not controllable. The subsequent columns refer to the version

Table 10 Percentage of failures and F-measure versus cluster size for hierarchical clustering with different linkage metrics for NanoXML

Cluster details		NanoXML version			
(% tests)		V1	V2	V3	V5
Count (%)	Size (%)	(%, F)	(%, F)	(%, F)	(%, F)
Single linkage					
1	50	***(64.28, 0.72)***	(0, 0)	(2.89, 0.03)	(0, 0)
5	11.95	(5.71, 0.09)	(5, 0.08)	(34.78, 0.5)	(40, 0.53)
10	6.37	(47.14, 0.59)	(40, 0.43)	(66.66, 0.79)	(41.53, 0.53)
15	5.03	***(64.28, 0.73)***	***(78.33, 0.84)***	***(82.60, 0.84)***	(60, 0.63)
20	3.31	(57.14, 0.65)	(68.33, 0.72)	(73.91, 0.75)	(60, 0.6)
25	2.76	(58.57, 0.62)	(68.33, 0.70)	(69.56, 0.70)	(44.61, 0.44)
Average linkage					
1	50	(0, 0)	***(100, 0.94)***	***(84.05, 0.91)***	***(100, 1)***
5	12.04	(7.14, 0.11)	(6.66, 0.11)	(14.49, 0.24)	(9.23, 0.14)
10	6.47	(44.28, 0.56)	(48.33, 0.64)	(34.78, 0.47)	(26.15, 0.35)
15	4.30	***(64.28, 0.73)***	***(78.33, 0.81)***	(68.11, ***0.74***)	***(60, 0.61)***
20	3.27	(64.28, 0.58)	(68.33, 0.61)	(***75.36***, 0.71)	(60, 0.56)
25	2.74	(58.57, 0.55)	(70, 0.57)	(69.56, 0.64)	(46.15, 0.37)
Complete linkage					
1	50	(0, 0)	***(100, 1)***	***(100, 0.98)***	***(100, 1)***
5	11.94	(0, 0)	(3.33, 0.004)	(26.08, 0.40)	(43.07, 0.60)
10	6.39	(2.85, 0.02)	(3.33, 0.03)	(49.27, 0.65))	***(70.76, 0.83)***
15	4.27	(31.42, 0.36)	(10, 0.08)	***(85.50, 0.88)***	(70.76, 0.74)
20	3.37	(52.85, 0.55)	(45, 0.45)	(76.81, 0.72)	(61.53, 0.60)
25	2.73	***(58.57, 0.57)***	***(70, 0.69)***	(69.56, 0.67)	(46.15, 0.42)

Table 11 Percentage of failures and F-measure versus cluster size for hierarchical clustering with different linkage metrics for Siena

Cluster details		Siena version		
(% tests)		V2	V4	V6
Count (%)	Size (%)	(%, F)	(%, F)	(%, F)
Single linkage				
1	19.8	(0, 0)	(0, 0)	(0, 0)
5	4	(17.85, 0.21)	(17.85, 0.15)	(17.85, 0.15)
10	1.99	(34.52, 0.31)	(34.52, 0.31)	(34.52, 0.31)
15	1.21	(61.90, 0.47)	(61.90, 0.47)	(61.90, 0.47)
20	0.79	***(75, 0.66)***	***(75, 0.66)***	***(75, 0.66)***
25	0.6	(60.71, 0.48)	(75, 0.62)	(75, 0.62)
Average Linkage:				
1	20.13	***(100, 0.96)***	***(100, 1)***	***(100, 1)***
5	4	(23.80, 0.19)	(23.80, 0.20)	(23.80, 0.20)
10	1.98	***(75, 0.65)***	***(75, 0.65)***	***(75, 0.65)***
15	1.20	(75, 0.57)	(75, 0.57)	(75, 0.57)
20	0.81	(71.42, 0.60)	(75, 0.62)	(75, 0.62)
25	0.6	(75, 0.64)	(75, 0.62)	(75, 0.62)
Complete Linkage:				
1	20	***(100, 1)***	***(100, 1)***	***(100, 1)***
5	4.04	***(100, 0.89)***	***(100, 0.89)***	***(100, 0.89)***
10	1.99	(60.71, 0.56)	(60.71, 0.56)	(60.71, 0.56)
15	1.27	(71.42, 0.52)	(71.42, 0.53)	(71.42, 0.53)
20	0.79	(75, 0.66)	(75, 0.66)	(75, 0.66)
25	0.6	(75, 0.62)	(75, 0.62)	(75, 0.62)

number of the programs and % and F refer to the percentage of failures and the F-measure. The figures in bold italics indicate the high values.

The data for NanoXML shows an interesting bi-modal response: the best results occur when there is either the smallest number of clusters (1 % which corresponds to two clusters) or when the cluster counts range between 10 and 25 % of the number of test cases (yielding between 20 and approximately 50 clusters). When the cluster count is very small, the algorithm will generate two large clusters (these will be of similar size but the smaller one is always treated as the small cluster) and in some cases one of these is composed entirely of failures and the other of passing outputs (those where the F-measure has a value of 1)—in other words the algorithm has managed to perfectly separate the passing and failing executions. These impressive clusterings were investigated in more detail and found that in version 2, 3 and 5 of NanoXML, all failing outputs follow exactly the same path through the program (despite being different faults generating distinct outputs) and the algorithms were perfectly separating the results based upon the execution trace. Even though this is probably a rare occurrence, it clearly demonstrates the power that execution traces can bring to this process.

As the cluster count increases so the results tend to drop quite dramatically until they pick up at around the 15 % level ($+/-5$ %) before tailing off again. In this range, the average small cluster sizes are between 2.73 and 6.39 % of the number of test cases—around 5 to 13 elements and it is worth noting that well over 60 % – sometimes far more—

Table 12 Percentage of failures and F-measure versus cluster size for hierarchical clustering with different linkage metrics for Sed

Cluster details		Sed version	
(% tests)		V5	
Count (%)	Size (%)	%	F
Single linkage			
1	19.8	24.2	0.3164
5	6.4	27.2	0.3265
10	2.6	16.6	0.2128
15	1.65	24.2	0.2402
20	1.2	**34.8**	**0.2872**
25	1	**34.8**	0.1839
Average linkage			
1	19.8	24.2	0.3164
5	7.6	13.6	0.2040
10	2.68	16.66	0.2053
15	1.67	25.7	0.2513
20	1.2	36.3	**0.2960**
25	1	**39.39**	0.2651
Complete linkage			
1	20	12.1	0.1948
5	6.6	22.7	0.2880
10	2.71	33.3	0.3757
15	1.69	28.0	0.2775
20	1.22	31.8	0.2589
25	1	**37.8**	**0.2933**

of the data points are failures. This again lends support to the experimental hypothesis behind this work that failures tend to congregate in small clusters. Another notable point is the fact that the F-measure tends to vary in line with the percentage of failures (and in all but one case the highest F-measure is also the highest percentage of failures), indicating that the failures associated with the numerous faults are evenly distributed across the small clusters. This is important as it could have been the case that the small clusters were dominated by a small and unrepresentative number of failures. The exact composition of these clusters will be explored in more detail later. It is also notable that both the linkage metrics and the versions of the program have an impact on the results, but the best overall and most stable results are produced by using the single linkage metric with a cluster count set at 15 % of the number of test cases.

The results for Siena (Table 11) tend to follow a similar pattern: in some cases the smallest number of clusters (5) tend to perform well and again manage to perfectly separate the data (once again this result is down to the passing and failing outputs being completely separable by their traces), but in other cases (with the single linkage metrics) they perform very poorly. The data for Siena also support the key hypothesis behind this paper with the cluster counts between 5 and 25 % of the number of test cases consisting of over 70 % failures. In contrast to NanoXML there is less of an impact of version (probably due to each having just a single failure) but like NanoXML the linkage metrics influence the

findings, with the single linkage producing the least consistent results and the complete linkage the best. The reasons behind this are unclear and need further investigation.

The picture for Sed is similar to that for experiment 1—a gradual increase in failure density and F-measure as the cluster size drops but a much lower overall failure density value than was observed in the other two projects. Including the trace information has not produced any dramatic results as with NanoXML and Siena as there is no dominant pattern of traces arising from failing executions.

The results of using EM and DBSCAN to perform the clustering are shown in Tables 13 and 14. The first column (systems) defines the subject programs with their version number. The second and third columns identify, as in the previous tables, the number of clusters and the average small cluster size again in terms of the percentage of test cases. The key difference in this case is that the cluster count is determined automatically by the algorithm. The final column shows the percentage of failures in the small clusters and the F-measure for each algorithm. With the exception of version 1, DBSCAN performed well on NanoXML: for version 2 the result was equal to the best found using agglomerative hierarchical clustering, and versions 3 and 5 were close to the best. It is also notable that the cluster count chosen was 15 %—identified as the best compromise for agglomerative hierarchical clustering. The trace information in version 1 is far more diverse which may explain the less impressive performance in this case. The results for Siena are consistent but far inferior to those produced by most of the different cluster size parameters using agglomerative hierarchical clustering. Sed produced the most disappointing results for this

Table 13 Percentage of failures and F-measure versus cluster size for DBSCAN clustering algorithm

Systems	Cluster details		DBSCAN
	Count (%)	Size (%)	(%, F)
Nanoxml V1	50	1.425	(32.85, 0.31)
Nanoxml V2	15	5.1	(78.33, 0.81)
Nanoxml V3	15	4.08	(79.71, 0.81)
Nanoxml V5	15	3.78	(69.23, 0.67)
Siena V2	6	3.33	(53.57, 0.48)
Siena V4	6	3.33	(53.57, 0.48)
Siena V6	6	3.33	(53.57, 0.48)
Sed V5	8	3.5	(9.0, 0.106)

Table 14 Percentage of failures and F-measure versus cluster size for EM clustering algorithm

Systems	Cluster details		EM
	Count (%)	Size (%)	(%, F)
Nanoxml V1	2.41	20	(40, 0.42)
Nanoxml V2	1.93	25.25	0
Nanoxml V3	2.41	19.8	(5.79, 0.09)
Nanoxml V5	1.44	33.33	0
Siena V2	1.41	14.28	0
Siena V4	1.41	14.28	0
Siena V6	1.41	14.28	0
Sed V5	1	33.33	(9.00, 0.085)

algorithm—far worse than when it was operating on test input and outputs alone which suggests that the clustering seems to be fragmenting the data further and is something that needs to be explored in future work. The findings for EM are very disappointing, with the odd exception of NanoXML Version 1. In the majority of cases, the algorithm failed to apportion any of the failures into the smallest clusters and also elected to use a very small number of clusters.

7.3 Fault density of smallest clusters

As in experiment 1, the practical utility of the approach and the return on investment was explored: how many outputs need to be examined before a reasonable number of failures and associated faults are observed? To answer this we examined in more detail the precise composition of failing data appearing in the smallest sized clusters—in other words which failing outputs appeared in which clusters.

The results of this analysis for NanoXML (with a clustering size of 15 % using agglomerative hierarchical clustering[7]) are shown in Table 15 (which takes the same form as Table 8 in Sect. 6.3). The three figures in the leftmost column show the absolute size of the cluster, the number of clusters of that size, and the size of the cluster proportional to the test set size (note that the table is presented in increasing order of cluster size and includes only clusters which are of less than average size). The second column identifies the failures found (indicated by 'Fn') and the number of occurrences of this failure. Failures associated with new faults (i.e. those not previously encountered) are indicated by a bold font. The final column is a cumulative count of the number of faults observed after examining the cluster over the total number of faults in the system. For example, the first entry of Table 15 shows that for Nanoxml Version 1 using 15 % of the number of test cases to define the number of clusters, there were ten clusters each of size 1 corresponding to 0.67 % of the number of test cases, containing failures 1 and 2 (2 times each) and 6 (once), giving a cumulative count of 3.

The NanoXML results show a number of failures appearing in the smallest clusters with additional ones appearing after examining just a few more clusters (with the exception of version 5). This is an important finding as it suggests that those failures which are going to be observed tend to appear relatively early in the ordering of clusters. This has important practical implications: collectively these smallest clusters correspond to between 25 and 30 % of the total output of the system, and the observed failures appear in an even smaller grouping, which means that the majority of failures in a system can be identified by looking at between one-fifth and one-quarter of the output—a substantial saving in effort for the developer.

The results for Siena are included in Table 16 although since Siena contains just the one fault the impact is less pronounced (and just one version is included since the results for other two are similar). However, it does show that the observed failures also tend to be concentrated early on in the small clusters and have the same implications as the NanoXML results.

The findings for Sed are shown in Table 17. The pattern is similar to the first experiment but the number of clusters to be examined has dropped very slightly. Again there are clear practical benefits: 75 % of the program's failures are concentrated in about 16 % of its results.

[7] This value was considered as it gave the best consistent performance over all versions of the system. Had the values varied between systems then some of the results would have been far more impressive.

Table 15 Failure distribution over less than average-sized clusters for NanoXML

Cluster size	Failures found	Cumulative
Version 1 (15 %)		
1, 10, 0.67 %	(**F1**:2, **F2**:2, **F6**:1)	3/7
2, 3, 1.34 %	(F1:2, F6:2)	3/7
3, 2, 2.01 %	(F2:3, F6:3)	3/7
4, 5, 2.68 %	(F2:4, **F5**:8, **F7**:8)	5/7
5, 1, 3.35 %	(F2:4)	5/7
6, 1, 4.02 %	(F2:6)	5/7
Version 2 (15 %)		
1, 8, 0.67 %	(**F1**:3, **F2**:2, **F6**:2)	3/7
2, 4, 1.34 %	(F1:2, F6:4)	3/7
3, 3, 2.01 %	(F2:3)	3/7
4, 5, 2.68 %	(F2:4, **F5**:8, **F7**:8)	5/7
5, 1, 3.35 %	(F2:5)	5/7
6, 1, 4.02 %	(F2:6)	5/7
Version 3 (15 %)		
1, 8, 0.59 %	(**F1**:3, **F2**:1, **F4**:2, **F6**:1)	4/7
2, 4, 1.18 %	(F1:2, F4:2, F6:2)	4/7
3, 5, 1.77 %	(F4:3, F6:6)	4/7
4, 6, 2.36 %	(F2:8, **F5**:8, **F7**:8)	6/7
5, 1, 2.95 %	(F5:5)	6/7
6, 1, 3.55 %	(F2:6)	6/7
Version 3 (15 %)		
1, 7, 0.62 %	(**F1**:1, **F2**:1)	2/8
2, 4, 1.25 %	(F1:2)	2/8
3, 3, 1.88 %	(-)	2/8
4, 6, 2.51 %	(F2:24)	2/8
5, 1, 3.14 %	(F1:5)	2/8
6, 1, 3.77 %	(F2:6)	2/8

Table 16 Failure distribution over less than average-sized clusters for Siena

Version 2 (5 %)		
Cluster size	Failures found	Cumulative
1, 5, 0.20 %	(–)	–/1
2, 1, 0.40 %	(**F**:2)	1/1
3, 8, 0.60 %	(F:24)	1/1
4, 1, 0.80 %	(F:4)	1/1
6, 3, 1.21 %	(F:12)	1/1
8, 1, 1.61 %	(–)	1/1
9, 1, 1.82 %	(F:9)	1/1
11, 3, 2.22 %	(F:33)	1/1

Table 17 Failure distribution over less than average-sized clusters for Sed version 5

Complete linkage (25 %)		
Cluster size	Failures found	Cumulative
1, 59, 0.19 %	(**F1**:8, **F2**:1, **F3**:7)	3/4
2, 17, 0.38 %	(F2:2, F3:4)	3/4
3, 6, 0.57 %	(F3:3)	3/4
4, 2, 0.76 %	(–)	3/4
5, 1, 0.96 %	(–)	3/4

7.4 Impact of failure density

One key factor in this study is the failure density. As mentioned in Sect. 5.1, this is between 31 and 39 % for NanoXML and 17 % for Siena. This failure rate is a factor of the combination of test cases supplied for the two systems and the nature of the faults embedded within the systems. However, in practical terms, this may be too high. The expectation is that this approach would be applied to a relatively mature system which may not have many obvious faults, and consequently a much smaller failure rate. Furthermore, an assumption behind anomaly detection is that anomalous events are relatively rare, whereas in these experiments the failure rate has been fairly high, so may represent a difficult case for the successful application of clustering techniques. To explore the impact of this, we took two versions of two of the systems—NanoXML V3 and Siena V4 (Sed was ignored as it demonstrated a similar failure rate to Siena)—and randomly pruned out fault revealing test cases to systematically reduce the failure rates to 10, 5 and 1 % for each system.

The results for this part of the investigation are shown in Tables 18 and 19 which, for each system, shows the cluster size, again in terms of the percentage of test cases (but note that the *actual* number of clusters will decrease as the failure rate decreases as test cases are being pruned from the suite), and the percentage of failures found and F-measure over the small clusters for failure rates of 10, 5 and 1 %. Both systems exhibit a similar distinctive pattern: as the failure rate decreases the recall (percentage of failures found) tends to remain high but the F-measure drops as the cluster count increases. The reason behind this is that with an increase in the number of clusters the false positive rate also increases as more passing tests become classified into the small clusters. This also has an important practical implication for this technique suggesting that if the system under investigation is expected to have a low failure rate, then the cluster count (if specified as a parameter) should be very small, but as the expected failure rate increases then so should the number of clusters. Further experimentation is required in order to validate this observation.

8 Threats to validity

The main threat to the validity of this study is the limited number and types of subject programs used in our experiments along with their associated faults and failure rates (although some investigation of the impact of reducing the failure rate has been undertaken). The input/output pairs of the subject programs were string data, and the programs themselves were of moderate size. The coding scheme also indicates a potential threat, but

Table 18 NanoXML V3 with reduced failure rate

NanoXML						
Cluster count (%)	10 %		5 %		1 %	
	Failures found (%)	F-measure	Failures found (%)	F-measure	Failures found (%)	F-measure
1	100	1	100	1	100	1
5	100	1	100	1	100	0.43
10	100	0.88	100	0.60	100	0.11
15	100	0.72	100	0.43	100	0.09
20	100	0.59	100	0.33	100	0.07
25	100	0.56	100	0.34	100	0.09

Table 19 Siena V4 with reduced failure rate

Siena						
Cluster count (%)	10 %		5 %		1 %	
	Failures found (%)	F-measure	Failures found (%)	F-measure	Failures found (%)	F-measure
1	100	1	100	0.94	85	0.91
5	100	0.78	0	0	100	0.14
10	100	0.69	100	0.43	100	0.13
15	52	0.43	100	0.4	100	0.14
20	100	0.64	100	0.42	100	0.11
25	100	0.58	100	0.31	100	0.07

this was created by examining a subset of inputs and outputs in ignorance of whether they are passing or failing pairs and then applied automatically to the remainder of the data set. This is relatively early work in this area, and the aim is to mitigate these threats by exploring a wider range of systems in the near future.

9 Conclusions and future work

This paper has presented an extension study of our preliminary study Rafig and Roper (2015) and investigated several clustering techniques such as agglomerative hierarchical, DBSCAN and EM clustering algorithms to build an automated test oracle. The input/output pairs investigated initially were augmented with execution traces with the aim of improving the proportion of unique failures in the smaller clusters.

The study confirmed the results of our earlier findings (Rafig and Roper 2015): in several cases small (less than average sized) clusters contained more than 60 % of failures (and often a substantially higher proportion). As well as having a higher failure density, they also contained a spread of failures in the cases where there were multiple faults in the programs. The results provide us with some useful guidelines in terms of specifying the number of clusters as a parameter to the algorithms. Over both experiments agglomerative

hierarchical clustering produced the most consistently good results, although performance varied according to which linkage metric was used (and also varied with experiment). The results for DBCAN were also generally encouraging, particularly since the number of clusters does not need to be supplied as a parameters.

The results also demonstrate important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining a relatively small proportion of the data to discover a large proportion of the failures. The approach has also been shown to be robust to a drop in the failure rate—all the way down to 1 % of the output—and initial results suggest that when the failure rate is likely to be low then the number of clusters should also be small.

Future research will be devoted to further empirical investigation of the effectiveness of our approach as an automated oracle, to corroborate the findings and to increase their external validity, particularly by exploring a wider range of programs and faults. Additional work includes exploring other anomaly detection strategies such as classification (mainly based on semi-supervised learning) with the aim of increasing the failure detection ability and reducing the false positive rate.

References

Baresi, L., & Young, M. (2001). Test oracles. Tech. rep.

Barr, E., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions*, *41*(5), 507–525. doi:10.1109/TSE.2014.2372785.

Bowring, J.F., Rehg, J.M., & Harrold, M.J. (2004). Active learning for automatic classification of software behavior. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04*, pp. 195–205. ACM, New York. doi:10.1145/1007512.1007539.

Briand, L. (2008). Novel applications of machine learning in software testing. In: *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pp. 3–10. doi:10.1109/QSIC.2008.29.

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Survey*, *41*(3), 15:1–15:58. doi:10.1145/1541880.1541882.

Dickinson, W., Leon, D., & Podgurski, A. (2001). Finding failures by cluster analysis of execution profiles. In: *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pp. 339–348. doi:10.1109/ICSE.2001.919107.

Dickinson, W., Leon, D., & Podgurski, A. (2001). Pursuing failure: The distribution of program failures in a profile space. In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9*, pp. 246–255. ACM, New York. doi:10.1145/503209.503243.

Do, H., Elbaum, S., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, *10*(4), 405–435. doi:10.1007/s10664-005-3861-2.

Doong, R. K., & Frankl, P. G. (1994). The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, *3*(2), 101–130. doi:10.1145/192218.192221.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., et al. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, *69*(1–3), 35–45. doi:10.1016/j.scico.2007.01.015.

Han, J., Kamber, M., Pei, J. (2012). 10-cluster analysis: Basic concepts and methods. In: J.H.M. Kamber, J. Pei (Eds.) Data Mining (Third Edition), The Morgan Kaufmann Series in Data Management Systems (third edition ed.), pp. 443–495. Morgan Kaufmann, Boston. doi:10.1016/B978-0-12-381479-1.00010-1. http://www.sciencedirect.com/science/article/pii/B9780123814791000101

Hangal, S., & Lam, M.S. (2002). Tracking down software bugs using automatic anomaly detection. In: *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pp. 291–301. ACM, New York. doi:10.1145/581339.581377.

ISTQB. (2016). ISTQB worldwide software testing practices report 2015–2016. Tech. rep. www.istqb.orgv

Jin, W., Orso, A., & Xie, T. (2010). Automated behavioral regression testing. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, vol. 0, pp. 137–146. doi:10.1109/ICST.2010.64.

Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of unix utilities. *Communications of the ACM*, *33*(12), 32–44. doi:10.1145/96267.96279.

Nguyen, C.D., Marchetto, A., & Tonella, P. (2013). Automated oracles: An empirical study on cost and effectiveness. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pp. 136–146. ACM, New York. doi:10.1145/2491411.2491434.

Pezzè, M., Zhang, C. (2015). Automated test oracles: A survey. In: *Advances in Computers*, vol. 95, pp. 1–48. Elsevier, Amsterdam.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., & Wang, B. (2003). Automated support for classifying software failure reports. In: *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pp. 465–475. IEEE Computer Society, Washington. http://dl.acm.org/citation.cfm?id=776816.776872

Podgurski, A., Masri, W., McCleese, Y., Wolff, F. G., & Yang, C. (1999). Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology*, *8*(3), 263–283. doi:10.1145/310663.310667.

Rafig, A., & Roper, M. (2015). Building test oracles by clustering failures. In: *IEEE/ACM 10th International Workshop on Automation of Software Test (AST 2015), ICSE (Supplemental Proceedings)*, vol. 2015, pp. 3–7.

Sekar, R., Bendre, M., Dhurjati, D., & Bollineni, P. (2001). A fast automaton-based method for detecting anomalous program behaviors. In: *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, pp. 144–155. doi:10.1109/SECPRI.2001.924295.

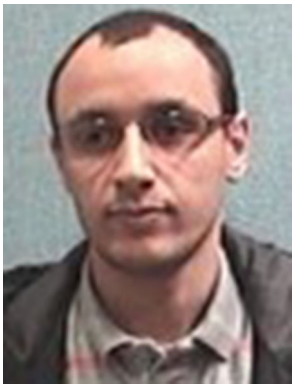
Vanmali, M., Last, M., & Kandel, A. (2002). Using a neural network in the software testing process. *International Journal of Intelligent Systems*, *17*(1), 45–62. doi:10.1002/int.1002.

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques*. Los Altos, CA: Morgan Kaufmann.

Yan, S., Chen, Z., Zhao, Z., Zhang, C., & Zhou, Y. (2010). A dynamic test cluster sampling strategy by leveraging execution spectra information. In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 147–154. doi:10.1109/ICST.2010.47

Yoo, S., Harman, M., Tonella, P., & Susi, A. (2009). Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*, pp. 201–211. ACM Press, New York.

Zhou, Z. Q., Zhang, S., Hagenbuchner, M., Tse, T. H., Kuo, F. C., & Chen, T. Y. (2012). Automated functional testing of online search services. *Software Testing, Verification and Reliability*, *22*(4), 221–243. doi:10.1002/stvr.437.



Rafig Almaghairbe is currently studying for his Ph.D. in the Department of Computer and Information Sciences at the University of Strathclyde. He obtained his B.Sc. degree in Computer Science from the University of Omar Al Mukhtar in 2006 and his M.Sc. in Computer and Internet Technologies from the University of Strathclyde in 2010. Previously, he worked as a lecturer assistant for several years after completing his B.Sc. in the Department of Computer Science at the University of Omar Al Mukhtar. His current research interests are in the development of automated support for building test oracles and the application of data mining.

Marc Roper is a Reader in the Department of Computer and Information Sciences at the University of Strathclyde. He obtained his B.Sc. (Hons) degree in Computer Science from the University of Reading in 1982 and his Ph.D. in Computer Science from the CNAA in 1988. His research has covered a number of areas of software engineering including design, development, and testing, and has typically incorporated significant empirical investigations: either based around controlled participant-based experiments or through the analysis of open-source systems and large-scale repositories. His current research interests are in the development of automated support for building test oracles and the application of data mining and search-based strategies for fault localisation.